AD-A219 943

(4)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AIM 1165	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Standard Map Machine		5. TYPE OF REPORT & PERIOD COVERED memorandum
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Brian LaMacchia and Jason Nieh		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE September 1989
		13. NUMBER OF PAGES 48
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC
S ELECTE APR 02 1990 D
B

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

chaos
numerical computations
special-purpose computing
nonlinear mappings
computer architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We have designed the Standard Map Machine (SMM) as an answer to the intensive computational requirements involved in the study of chaotic behavior in nonlinear systems. The high-speed and high-precision performance of this computer is due to its simple architecture specialized to the numerical computations required of nonlinear systems. In this report, we discuss the design and implementation of this special-purpose machine.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A.I. Memo No. 1165 September, 1989

The Standard Map Machine

by
Brian LaMacchia and Jason Nieh

Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

ABSTRACT

We have designed the Standard Map Machine (SMM) as an answer to the intensive computational requirements involved in the study of chaotic behavior in nonlinear systems. The high-speed and high-precision performance of this computer is due to its simple architecture specialized to the numerical computations required of nonlinear systems. In this report, we discuss the design and implementation of this special-purpose machine.

Keywords:
chaos, nonlinear mappings, numerical computations,
computer architecture, special-purpose computing

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Acknowledgements

We are grateful to Professor Gerald J. Sussman for providing us with the opportunity to learn. He first suggested the idea of special-purpose computing for dynamics and provided guidance concerning the applications of SMM and the methods of computation. Many thanks for all of his advice and encouragement.

Special thanks to Henry Wu for his help with the initial design of SMM and with the implementation of the machine. He provided valuable suggestions on many of the issues which arise in high-speed digital design. Henry also created some of the software tools which made debugging the SMM hardware much easier.

Much gratitude is due to Andrew Berlin for his suggestions and encouragement. He suggested a number of useful considerations in the initial construction of the machine and also helped debug the hardware.

Thanks also to Panayotis Skordos and Gerald Roylance for suggestions on numerical methods, Mike Eisenberg and Professor Jack Wisdom for explanations about nonlinear phenomena, William McAllister for suggestions about hardware debugging, Bill Rozas for helping with the final stages of debugging the system, Professor Harold Abelson for helpful comments in writing this report, and the other members of Project MAC for all their support and encouragement.

Contents

List of Figures

1 Introduction

High-speed computation is playing an ever increasing role in the pursuit of scientific endeavors. However, large areas of scientific research remain as yet inaccessable due to insufficient computing resources. There is demand in a wide variety of applications for faster and faster computation tools.

A technique that has proven effective in tackling such computationally intensive problems is special-purpose computing. Special-purpose computing has been used in a number of different areas with much success. For instance, the Digital Orrery, designed to simulate the long-term behavior of the solar system, performed the longest integration of the orbits of the planets to produce evidence that the motion of Pluto is chaotic. We see the power of special-purpose computing in competitive chess, where the computer Deep Thought has defeated numerous high-ranking chess masters, as well as chess programs running on general-purpose supercomputers.

The performance advantage of special-purpose machines is derived from a number of factors. The application-specific design allows more highly optimized architectures of less complexity than their general-purpose counterparts. These simpler machines are more cost-effective and can be dedicated solely to a particular problem, rather than having to be shared, as is the case for large expensive supercomputers.

In this light, we have designed the Standard Map Machine (SMM) as an answer to the intensive computational requirements involved in the study of chaotic behavior in a class of nonlinear systems. SMM is tailored to perform nonlinear mappings with high speed and high precision. The prototype implementation fits on a single board and performs at an average rate of about 2.5 MFlops for the class of problems that it was designed to solve. The novel computer design and implementation are presented in the following sections.

3 Architectural Design

Figure 1 shows the architectural configuration of SMM. The machine has two sections: the computation unit and the microcontroller. Data path specialization and instruction pipelining result in maximum utilization of the multiplier/ALU floating point module.

3.1 The Computation Unit

The computation unit has four parts: the data memory, the register file, the multiplier/ALU module, and the feed-through latch. These parts are interconnected by 32-bit data paths. Data is moved between SMM and the host computer via an interface to the data-memory system. Values that are to be used for a computation are written to the fast dual-port register file and then fed into the multiplier/ALU unit. The multiplier/ALU component is made up of a Weitek 1264 64-bit IEEE floating point multiplier and a Weitek 1265 64-bit IEEE floating point ALU. Normally, the floating point chips are operated in pipeline mode to maximize throughput. In this mode, a new 64-bit multiplier operation can be initiated every four clock cycles with a latency of ten clock cycles, while a new 64-bit ALU operation can be started every two clock cycles with a latency of twelve clock cycles. (The minimum clock cycle time for the Weitek chips is 60 nanoseconds.) The results of floating point operations are then sent to the data memory, register file, feed-through latch, or any combination of the three. Intermediate results are ususally stored in the register file. The data memory is used if there is no free space left in the register file or if the value is not needed immediately. If the result is to be used as an input to the multiplier/ALU module immediately after it has been calculated, it is fed directly to the feed-through latch. The result must be written to data memory if it is to be sent to the host computer.

The feed-through latch is an additional data path feature developed to optimize the computation of polynomial approximations of special functions such as sine and cosine. From the series expansion in Equation 3, we see that such functions are computed serially for a small number of execution units.[1] Because of this, the computation of each successive iteration in mappings such as the standard map is a serial computation, since the calculation of both x and y values depends on the value of the sine function. The feed-through latch takes advantage of the serial

[1]If there are a large number of execution units, the polynomial can be rewritten so that the terms can be calculated in parallel, independent of each other.

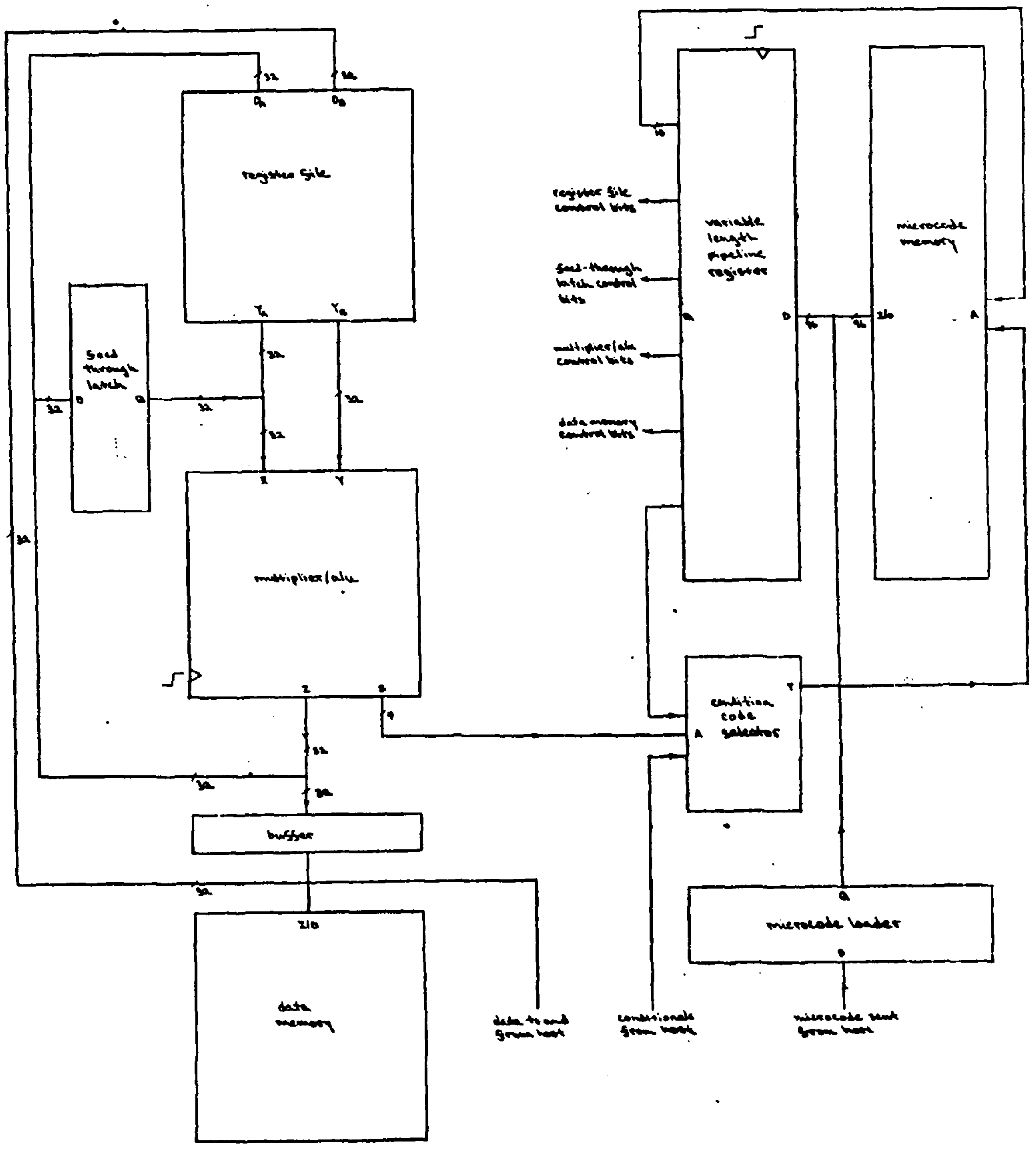


Figure 1: Architectural Description

nature of the calculation by allowing the results of computations from the floating point unit to be fed directly back into the inputs to the floating point unit without passing through the register file. Data values can remain in the data paths for repeated computations without ever having to be written back to the register file, thus reducing the latency of the computation.

Besides numerical data calculations, the Weitek unit is also used to perform conditional tests, such as comparing the values of two numbers, and converting floating point numbers to integers for data memory write address calculations. Write addresses are calculated such that the low eight bits of the resulting integer are sent to the memory address register and latched in as the high eight bits of a memory write address. This allows fast calculation of page addresses to which to write the results of data computations. Furthermore, the amount of data memory that can be used to write results is not limited by the microcode memory size, as would be the case if write addresses were not calculated but rather were fixed by the microcode instruction. Data memory read addresses are fixed by the microcode instruction because much of the data memory, where the final results of computations are stored until being sent to the host, will not need to be read for the kinds of applications that SMM will run, and direct addressing provides better latency so that data can be read immediately upon execution of the read instruction.

3.2 The Microcontroller

The microcontroller consists of a microcode loader, the microcode memory, a variable-length pipeline register, and a condition code selector. Programs for the computer are assembled into microinstructions on the host machine and downloaded from the host through the microcode loader, which then stores the microinstructions into the micromemory. When the microcontroller is operating, the program stored in the microcode memory is executed. The instruction located at the current microaddress is read from the memory and sent to the variable-length pipeline register. The instruction is clocked in through the register, resulting in the proper control signals being sent to the computation unit as well as the microaddress of the next instruction being sent to microcode memory. The length of the pipeline is different for each control signal so that each of the signals coming out of the microcontroller arrives at the appropriate module at the right time. In addition, the length of the entire instruction pipeline is varied by the microinstruction, depending upon whether the multiplier or ALU is being activated. This variation is necessary because the pipelined latency of the multiplier differs from that of the ALU. The condition code selector sends branching instructions to the microcode memory depending on the current state of the machine and the result of conditionals computed by the computation unit. Thus, data dependent instructions are permitted.

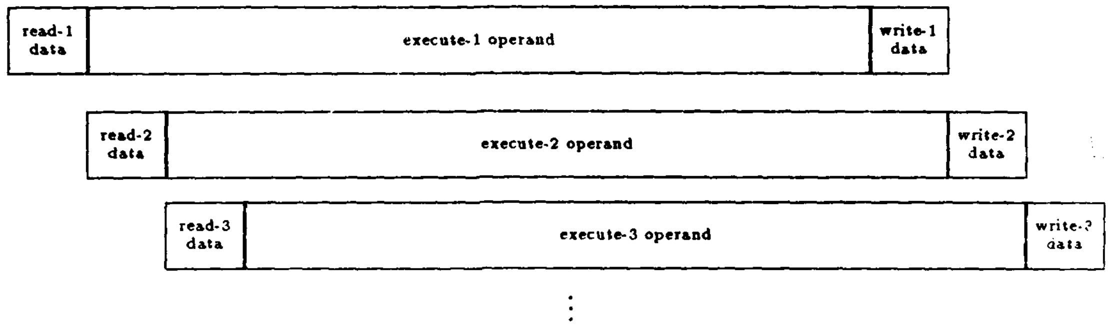


Figure 2: Model of Instruction Flow

A 96-bit instruction word length was chosen. The function of each instruction bit is described in Appendix A. The long word instruction format allows parallel execution of the subsystems of the computation unit. Instruction flow is as shown in Figure 2. One instruction word, and thus one single-precision floating point operation, can be executed every clock cycle. This performance is possible because of the variable-length hardware pipeline register. As mentioned above, the microcode dynamically alters the length of the pipeline so that the control signals arrive to the appropriate subsystem modules at exactly the right time. Furthermore, the hardware pipelining easily facilitates multiple branching by guaranteeing that the write address for the result of the data operation will not disappear until the result has been written, despite the interleaving of instructions.

Each instruction word can be described functionally as shown in Figure 3. The memory field is used to write a data value stored in data memory to the register file. The computation field is used to perform an operation using the multiplier/ALU module. The branching field is used for conditional branching.

4 Implementation

Our implementation of SMM uses standard off-the shelf parts, primarily Advanced Schottky TTL technology. Single-port SRAM is used for both the data and microcode memory systems. There are 2 K words of microcode instruction memory, 8 K words of data memory, and 64 words of register file locations, where an instruction word is 96 bits in length and a data word is 32 bits in length. Four-level pipeline registers that combinatorially select which pipeline level the resulting output comes from are used to implement the microcontroller pipeline. The system clock, write pulses, and latch pulses are all derived using a single delay line. Schematics are given in Appendix B.

4.1 Timing

Timing analysis of the machine is shown in Appendix B. Positive edge timing is used. Maximum and minimum propagation delays are taken into account, as well as the tolerances on each tap of the delay line. The minimum clock cycle time is 62.5 nanoseconds as limited by the critical data path latency from the output of the Weitek unit through the feed-through latch back to the input of the Weitek unit. As a result, a 70 nanosecond delay line was chosen for the initial implementation. Data reads are done during the first section of the clock cycle and data writes are done on the later section of the cycle.

4.2 Construction

The computer was built on a 366 mm x 220 mm wire-wrap board with ground and voltage planes to reduce noise problems. A picture of the completed hardware

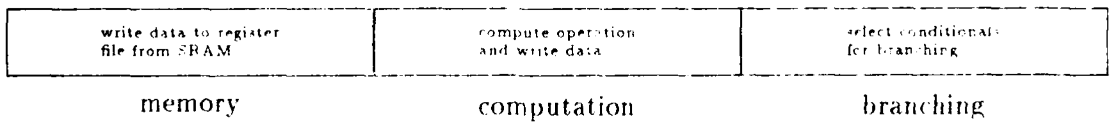

write data to register file from SRAM
compute operation and write data
select conditionals for branching
memory
computation
branching

Figure 3: Functionality of an Instruction Word

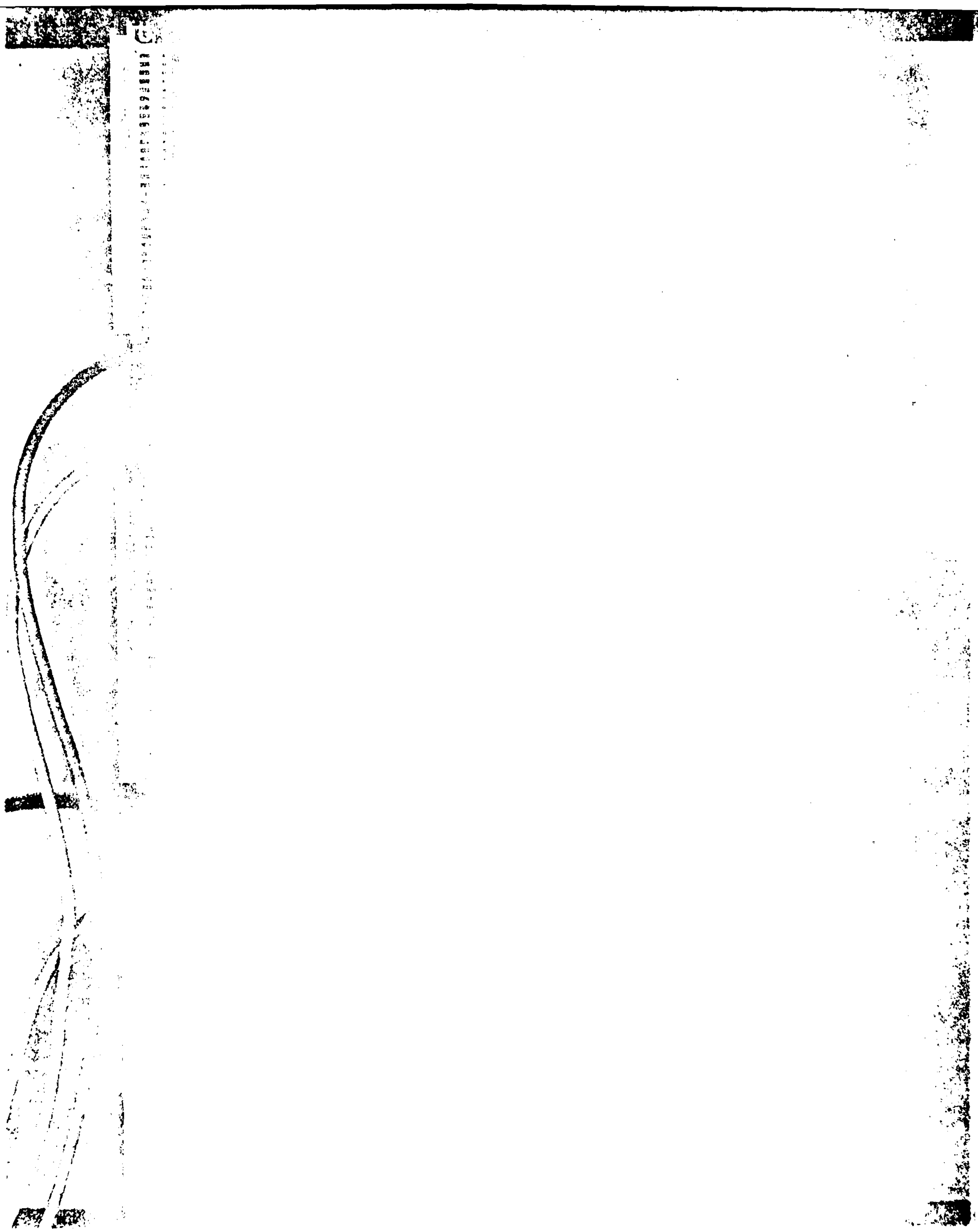
is shown in Figure 4. $.1\mu F$ bypass capacitors were soldered directly between Vcc and GND on dual-in-line chip packages and $47\mu F$ electrolytic bypass capacitors were distributed all over the board. In addition, $4.7\mu F$ tantalum capacitors were connected between Vcc and GND near the register files and Weitek chips because of their extra sensitivity to noise. Clock signals were fed through multiple buffers to avoid fanout and ground bounce problems. Twisted pairs were used for some of the longer wires to reduce noise. Several of the wires required termination because of heavy undershoot resulting from transmission line bounce [6]. Some of the wires connected to CMOS inputs were also attached to TTL inputs to reduce undershoot by taking advantage of TTL input clamping diodes. The current implementation uses a 100 nanosecond delay line to produce the system clock, resulting in a 100 nanosecond clock cycle time. Some minor adjustments of the wiring that have not yet been done at this time would reduce the transmission line noise that is present and allow a faster cycle time.

4.3 Host Interface

The Standard Map Machine was implemented to communicate with an HP 9000 Series 300 computer system [5]. The HP 98630 Breadboard Interface was used to construct memory map hardware to communicate between the backend processor and the host machine. The asynchronous nature of the protocol requires that the processor be stopped or in a "waiting state" in order for data to be transmitted to and from the processor and the host.

The SMM "waiting state" is a special state the machine can enter while the system clock is running. While in this state, SMM promises not to access the data memory so that the host machine can access the memory safely without worry of bus contention. It is possible to instruct SMM to enter and exit waiting states from software, thus allowing data to be sent to and from the host during the execution of a program.

Another way that the host can read and write data to SMM is to send an interrupt signal to SMM, stopping the system clock. In this state, the machine is no longer running and data can be transmitted between SMM and the host. In addition, programs can be loaded into the microcode memory since the microcontroller has stopped running. Note that programs cannot be loaded if SMM is running (i.e. the system clock is on). To start SMM, a "start" signal is sent to SMM, which sends the starting instruction from the microcode loader to the instruction register and enables the SMM clock.



Software was written to allow the user to interface with SMM in Scheme. Commands that are available to the user are listed in the Appendix D. These commands give the user great flexibility in the operation of the machine as well as easy access to the data memory system where the results of computations are stored.

5 Programming SMM

This section describes the programming language we created for SMM and the assembler which converts that language into SMM microinstructions.

5.1 The SMM Language

We designed a *register transfer language* (RTL) for writing SMM programs. The specific syntax of this language is detailed in Appendix C. Every RTL instruction has three mandatory parts: an instruction opcode, a data source, and a data destination. Optionally, an instruction specifies a conditional or unconditional branch in a fourth clause.

Although RTL is similar to actual SMM microinstructions, it is sufficiently abstracted from the SMM hardware that programs may be written quickly. For example, instruction macros allow programmers to specify double-precision operations in one RTL instruction, even though it takes at least two cycles for SMM to move double-precision numbers around. The expansion of double-precision macros is one of the many tasks handled by the assembler.

The choice of the RTL model is well suited to SMM. All fields of an SMM instruction are issued simultaneously; hardware pipelining ensures that control signals arrive at each pipe stage during the appropriate cycle. This approach avoids the overhead of flushing the pipeline before branches and the complexity of implementing software pipelines in the assembler. Furthermore, the instruction pipeline allows the programmer to completely specify an instruction at the beginning of the operation, rather than having to manually follow data through SMM[2]. The instruction pipeline allows RTL programs to be written for SMM quickly and efficiently.

[2]Note that programmers still have to be aware of how long an operation takes so as to not reference a data value before it has been computed.

5.2 The Assembler

Chasm, the CHaos ASseMbler, is our assembler for translating the SMM register transfer language into microcode. *Chasm* is a multi-pass assembler; each phase performs a piece of the assembly operation on the entire instruction stream until the desired instruction bits are obtained. Below we detail the five *chasm* assembly phases.

5.2.1 Phase 1: Macro expansion

Phase 1 of *chasm* is responsible for reading the source code list and expanding all instruction macros. Macros come in two flavors: math macros and instruction macros. Math operation macros allow the programmer to use simplified expressions like `(F->I <FOO>)` (float-to-int convert <FOO>) instead of the fully specified operation `(F2I <FOO> (NONE))`. Instruction macros expand one instruction into many, such as the `(WAIT)` loop macro and the `DA` (double-precision assign) opcode. The output of *chasm* Phase 1 is a stream of valid, one-cycle instructions.

When Phase 1 encounters a double-precision instruction macro, it replaces the macro with two single-precision instructions. The first instruction uses the same source and destination as the original instruction, but modifies the math operation (if any) to tell the multiplier/ALU that this is the beginning of a double-precision instruction. The second instruction uses sources and destinations at addresses *one greater* than the original instruction. This allows easy access to 64-bit quantities by using double-precision instructions with the addresses of the low 32-bit words. For example, the instruction:

```
(DA (R 40) (+ (R 10) (R 20)))
;; Assign the sum of quantities in (R20,R21) and (R10,R11) to (R40,R41)
```

expands into the two instructions:

```
(A (R 40) (D1+ (R 10) (R 20)))
(A (R 41) (D2+ (R 11) (R 21)))
```

Here `D1+` and `D2+` perform the double-precision addition function.

5.2.2 Phase 2: Instruction Placement

Phase 2 of the assembler is responsible for assigning to each macro-expanded instruction a location in the SMM microcode memory. Generally, instructions are placed sequentially starting at microcode address 0. Difficulties arise, however, when placing conditional instructions.

Conditional branches in SMM are implemented by substituting one of seven one-bit condition codes for the least significant bit (lsb) of the microcode address. Thus, when we place a conditional instruction, we must make certain that:

1. The *next* instruction (i.e. "condition false, proceed to next sequential instruction") in the instruction stream must be located at an even address (lsb = 0).

2. The instruction immediately *after* the next sequential instruction in the stream (lsb = 1) must be a *copy* of the instruction the machine is supposed to jump to.

Phase 2 also performs two other important tasks. First, it makes sure that instructions are properly "linked" so that execution proceeds properly from one instruction to the next, regardless of where those instructions are located in memory. Second, this phase places a `(JUMP 0)` instruction at the end of the code block. Since a copy of the last instruction downloaded remains in the instruction latches, this forces execution to begin at the start of the program.

5.2.3 Phase 3: Label Collection

The RTL allows any instruction to have a *label* for referencing by other instructions. Labels are useful for designating the destinations of `IF` clauses and `JUMP` instructions. *Chasm* Phase 3 associates an instruction address with each instruction label present in the source code. These associations are used later when calculating the instruction addresses of `(GOTO :label)` clauses.

5.2.4 Phase 4: Instruction Duplication

In Phase 2, *chasm* copied instructions that are destinations of conditional jumps by marking the high-order instruction with a `COPY` token along with the label of the instruction to be copied. Now that Phase 3 has collected label information and calculated instruction linkage information, these tokens may be replaced by actual instructions. *Chasm* Phase 4 copies instructions as necessary, making sure that copied instructions link execution to the same instruction as the original. By the end of Phase 4, we are ready to build the bit strings to be downloaded to SMM.

5.2.5 Phase 5: Bit String Construction

Phase 5 of *chasm* converts symbolic RTL into the 96-bit strings to be downloaded into SMM. For each instruction, Phase 5 initializes a `NOP` bit string[3] and then proceeds to modify that string based on the contents of the instruction. The output of Phase 5, the **`*chasm-bits*`** vector, is a vector of bit strings which may be downloaded (by the `download-code` procedure) directly into SMM for execution.

6 Performance

The inherent maximum performance of any machine using one Weitek 1264/65 chip set is approximately 4 MFlops for multiplier operations, clocking the machine at the minimum clock cycle time of 60 nanoseconds. SMM approximately achieves this absolute maximum when clocked at its maximum rate of 16 MHz. Actual performance of this computer for the class of problems it was designed to solve is limited by the throughput of the multiplier due to the serial nature of the computation. The current hardware is running at a slower clock rate of 10 MHz, thus yielding about 2.5 MFlops for the single board machine. Since the ratio of time spent in host-processor intervention versus that in numerical calculation is small, these performance levels are sustainable for long periods of time.

This high performance is achieved by specializing the data paths for numerical computations and the serial nature of the problems that the computer was designed to solve. Furthermore, variable-length hardware pipelining maximizes the throughput of the machine by encouraging interleaved instructions. The major advantage in the SMM design is that it allows the user to express large portions of real problems as efficient sequences of microcode instructions. These instructions perform large amounts of computation without ever requiring the host to manage control and data flow on the board.

[3]That is, a bit string which has the effect of a **`No OPeration`** instruction.

The pipelined operation of SMM allows simultaneous calculation of up to three mappings in problems like the standard map. This form of parallelism is well suited to the study of the behavior of two-dimensional mappings since we can simultaneously integrate three nearby trajectories and monitor their divergence.

7 Conclusion

Studying chaotic behavior in nonlinear systems through simulation requires numerous calculations. The Standard Map Machine is a special computer designed and implemented to perform these intensive calculations, such as the computation of the standard map for millions of iterations. The prototype implementation fits on a small wire-wrap board and provides 2.5 MFlops of double-precision computing power for the class of problems that it was designed to solve. Its high-speed and high-precision performance are due to the specialization of its architecture to the numerical computations required of nonlinear systems.

This backend computer has numerous advantages over conventional floating point accelerators and math coprocessors. Almost all of the computations can be performed on SMM itself using its own fast microcontroller, rather than relying on the slower instruction control of the host machine. This reduces the communication costs between the host and the backend processor, a factor that heavily reduces the performance of other machines. Furthermore, unlike costly supercomputing power, SMM serves as a cost-effective instrument that can be completely dedicated for long periods of time to numerical simulations of nonlinear systems.

As technology improves, we claim that machines of a similar nature can be designed and implemented as effective instruments for scientific computing. Current chip technology can already provide at least twice the performance of the floating point multiplier/ALU chip set used in SMM. The simple architecture of special-purpose computers allows numerical operations to be implemented more efficiently than is feasible on a general-purpose machine. Specialized numerical architectures provide the edge in high-speed and high-precision performance necessary for intensive computations.

A Control Bits of the SMM Instruction Word

control bits	functional description of bits	possible values for control field
0 - 9	microaddress bits 1-10	0 - 1023
10	wait	0 for not waiting, 1 for waiting
11	regfile B-port write-enable	0 for not enable, 1 for enable
12	regfile A-port latch-enable	0 for not enable, 1 for enable
13	regfile B-port latch-enable	0 for not enable, 1 for enable
14 - 19	regfile A-port read-address	0 - 63
20	regfile A-port/latch output	0 for regfile, 1 for latch
21 - 26	regfile B-port write-address	0 - 63
27 - 32	regfile B-port read-address	0 - 63
33	regfile B-port output	0 for enable, 1 for not enable
34 - 39	1264 multipler function	0 - 63, see Weitek spec.
40 - 45	1265 ALU function	0 - 63, see Weitek spec.
46	1264 load enable	0 for enable, 1 for not enable
47	1265 load enable	0 for enable, 1 for not enable
48 - 52	1264 load control	0 - 31, see Weitek spec.
53 - 57	1265 load control	0 - 31, see Weitek spec.
58 - 60	1265 unload control	0 - 7, see Weitek spec.
61	multiplier/ALU select	0 for ALU, 1 for multipler
62	regfile A-port write-enable	0 for not enable, 1 for enable
63	feed-through latch pulse	0 for do not latch, 1 for latch
64	SRAM write-enable	0 for not enable, 1 for enable
65	SMAR address latch pulse	0 for not enable, 1 for enable
66 - 71	regfile A-port write-address	0 - 63
72	Weitek-to-SRAM buffer	0 for enable, 1 for not enable
73 - 75	conditon code select A, B, C	select cc0 - cc7
76 - 78	1264 unload control	0 - 7, see Weitek spec.
79 - 90	SRAM address	0 - 4095
91	default condition code	0 or 1, set by assembler
92	condition code latch pulse	0 for not enable, 1 for enable
93 - 95	unused	

Table 1: Description of SMM Instruction Bits

B Design Schematics and Timing Diagrams

The design schematics and timing diagrams for SMM and the host interface are shown in the following figures. Figure 5 gives a functional description of the overall system with a detailed view of the system clock design showing the clock generator as well as the derived write pulses and latch pulses. Note that the clock buffers that were included in the actual construction to reduce fanout are not shown in the diagram. Figure 6 gives a detailed diagram of the wire connections in the computation unit. Figure 7 is a detailed drawing of the microcontroller. There are three bits of the 96-bit instruction word that are currently unused. Figure 8 shows the hardware for the host interface as well as the timing of interface signals. The signals from the host are transmitted from the breadboard interface card that is plugged into the backplane of the host, through ribbon cables, to the control and data buffers. The breadboard interface is not shown here. Figure 9 shows the timing analysis of the system.

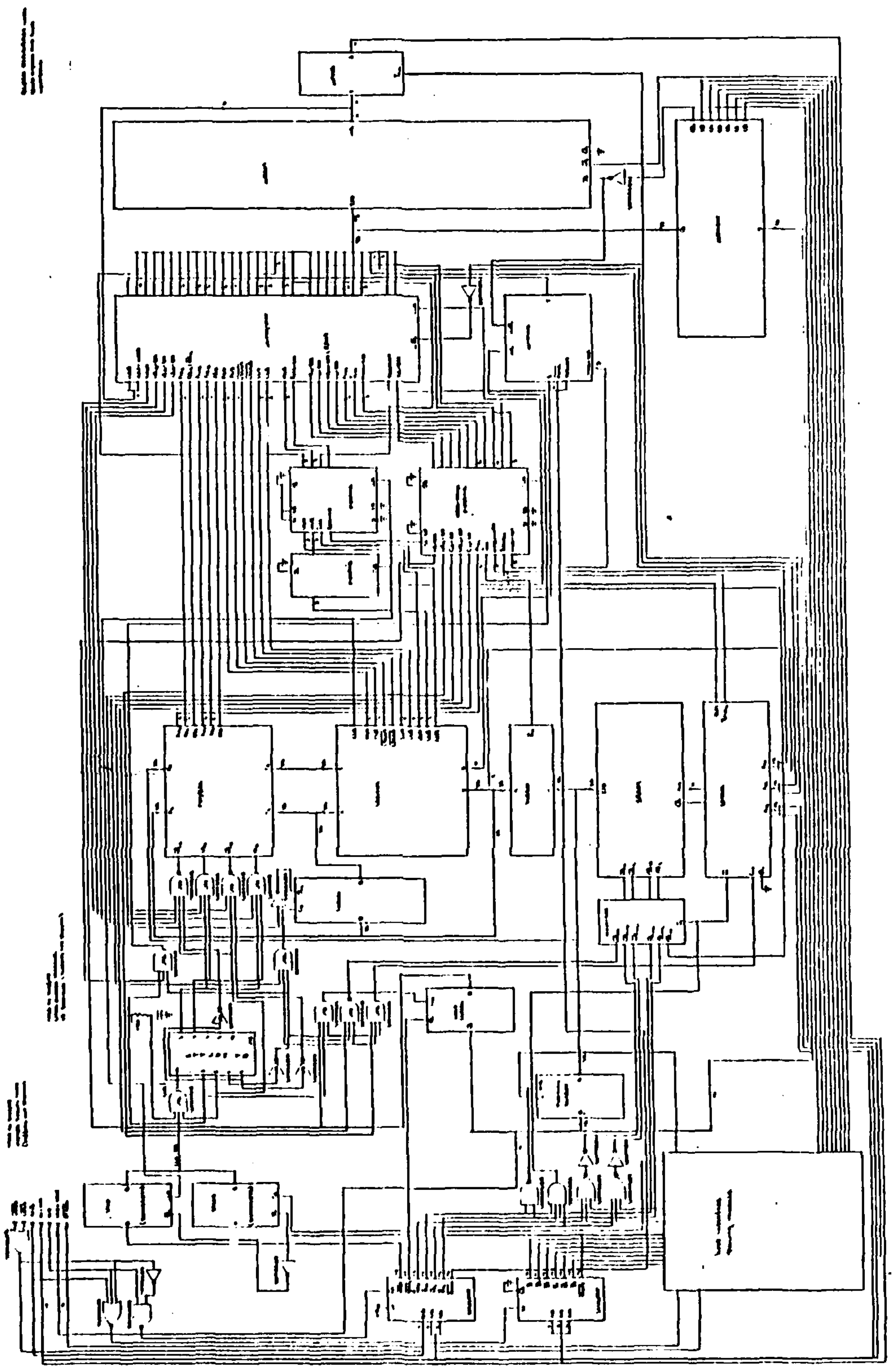


Figure 5: Functional Description with System Clock

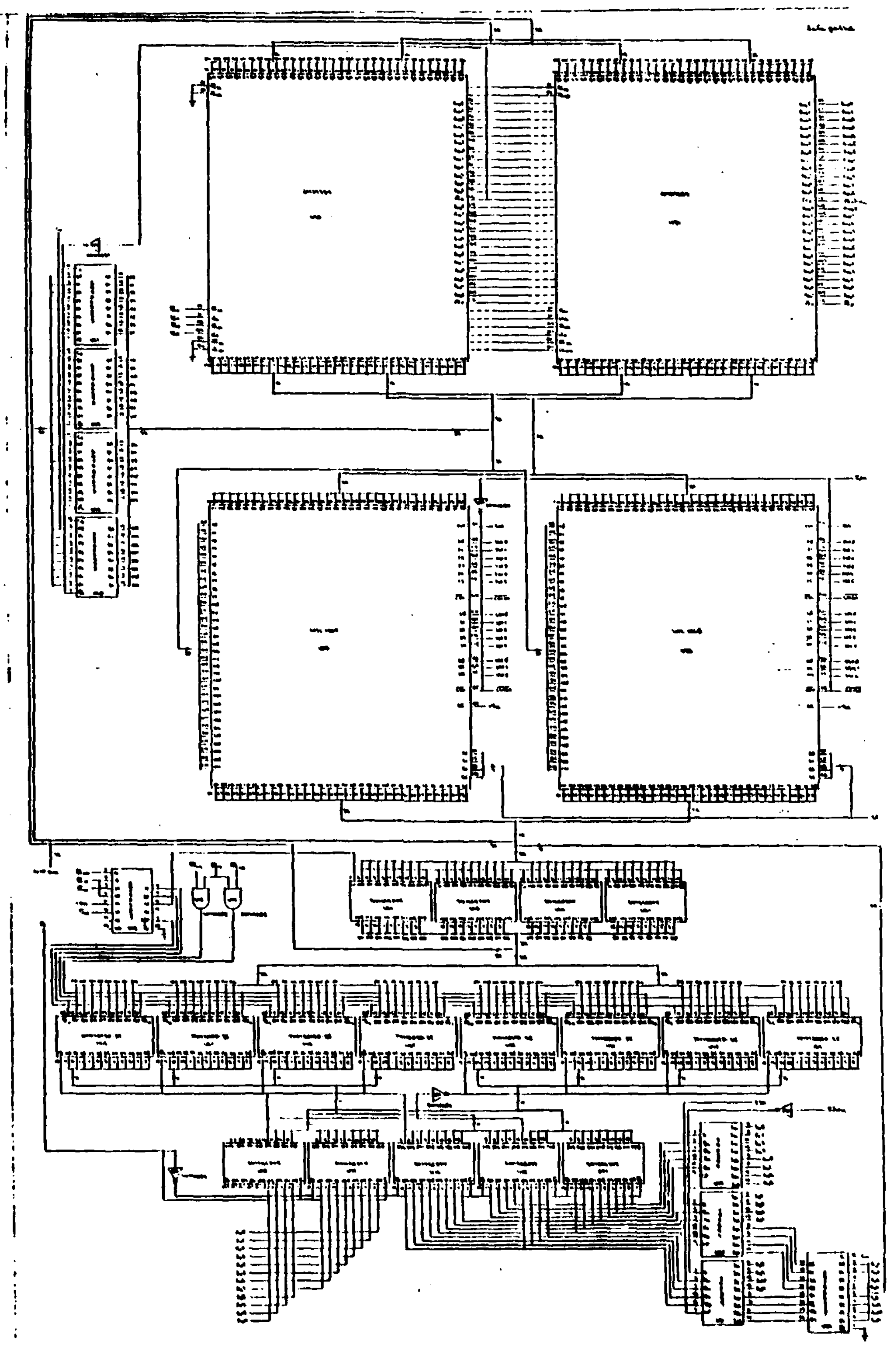


Figure 6: Computation Unit

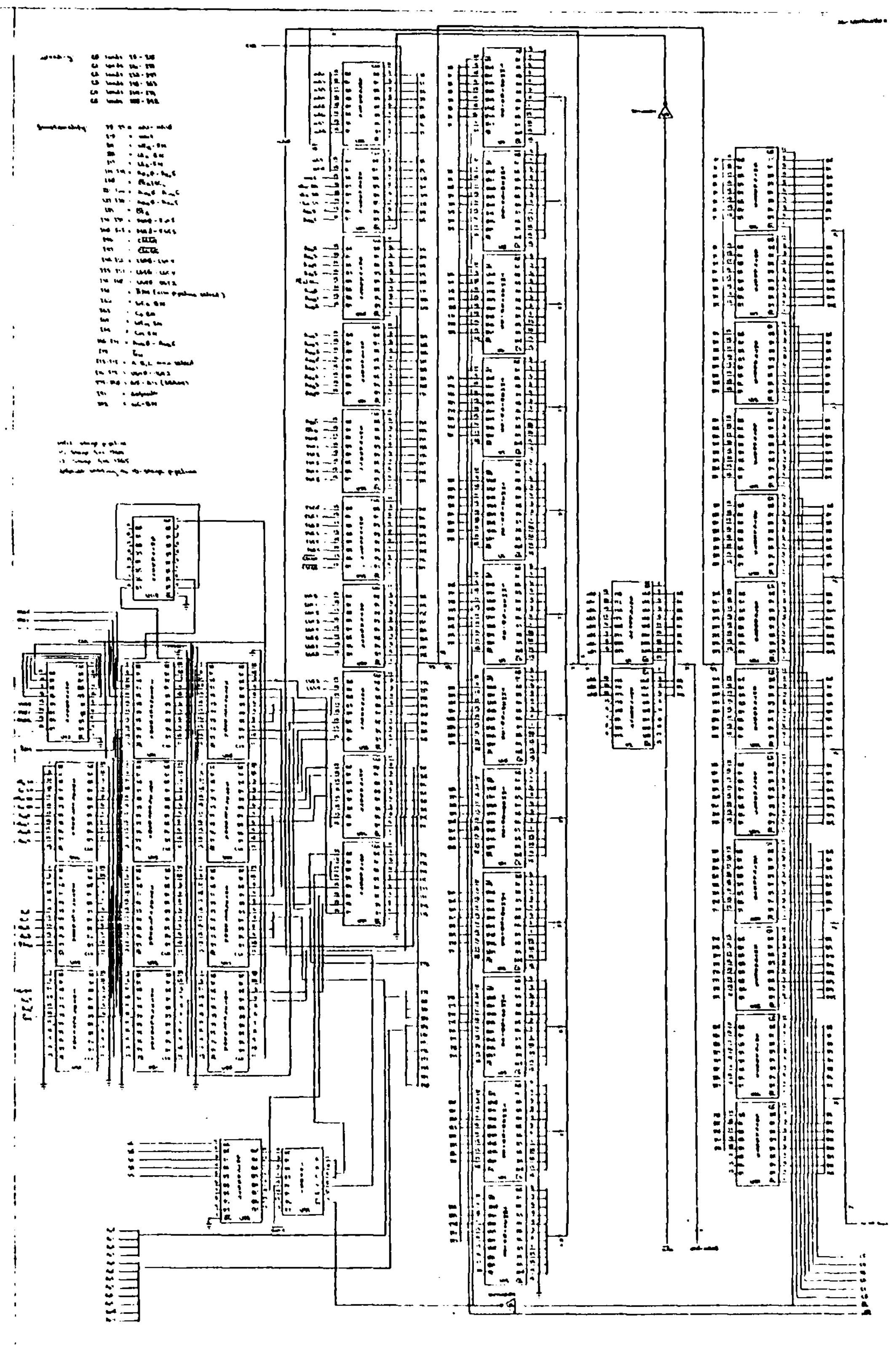


Figure 7: Microcontroller

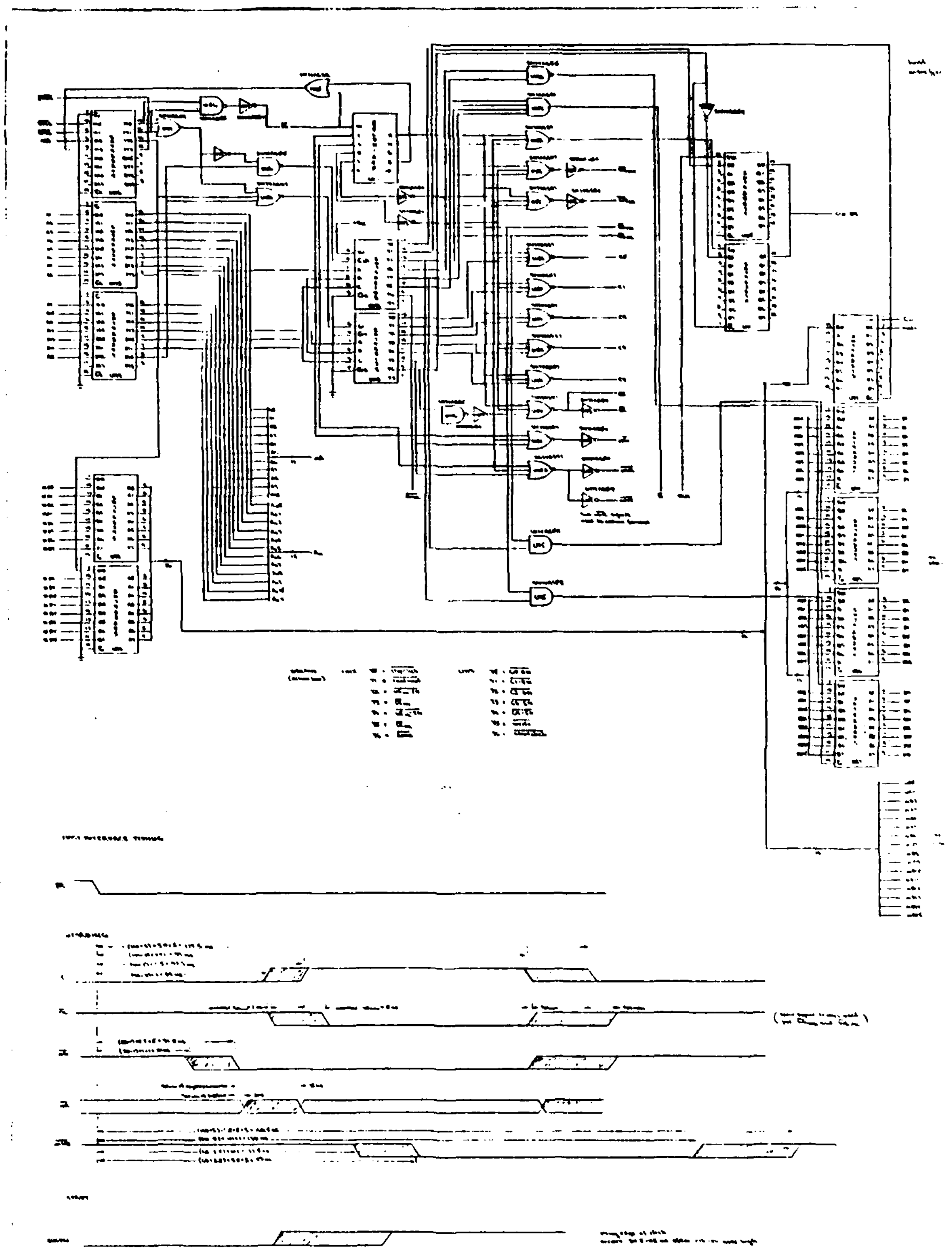


Figure 8: Host Interface

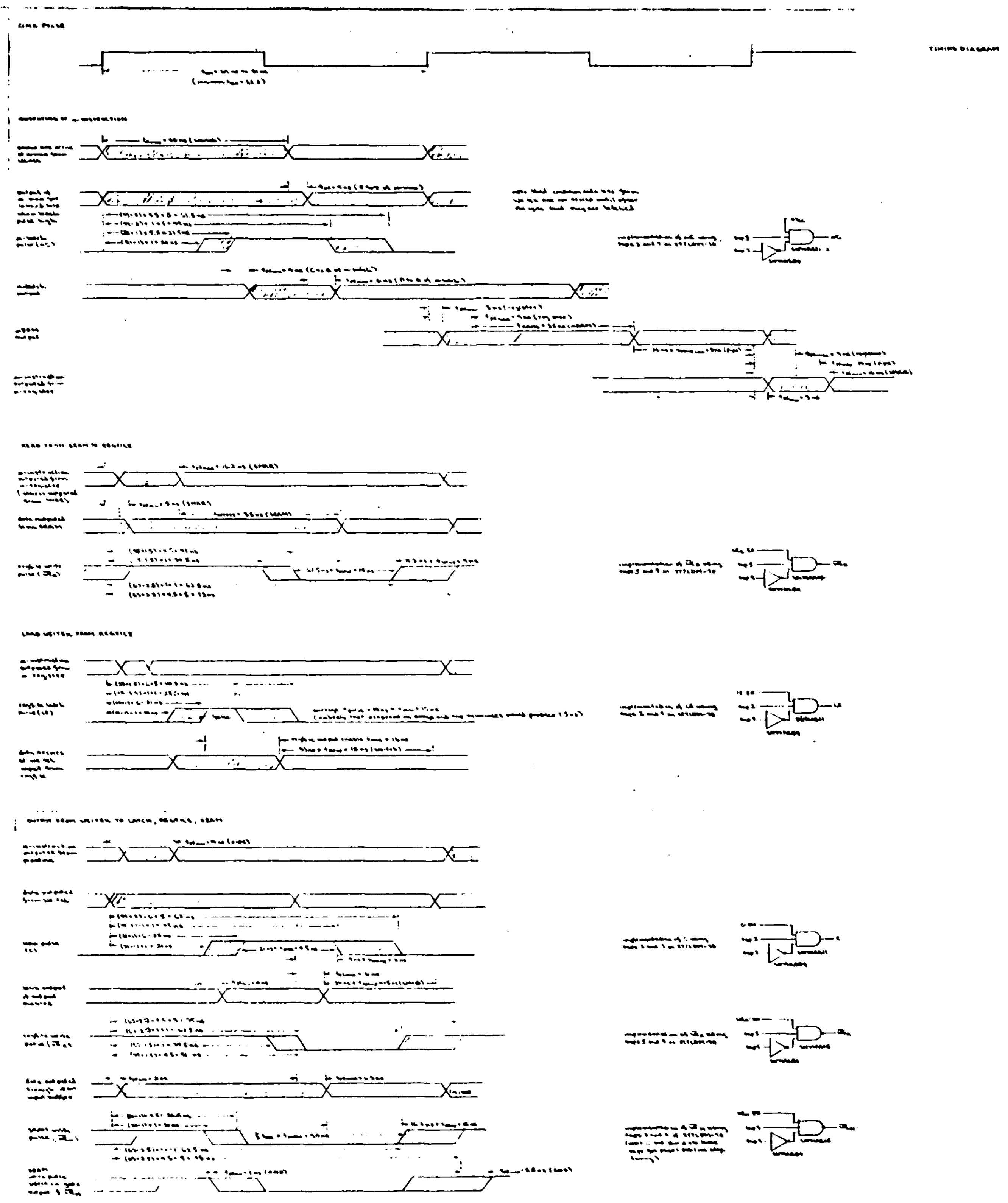


Figure 9: Timing Diagram

C The Register Transfer Language

This section describes the instruction set of the SMM register transfer language. We first present a symbolic description of the language, and then describe the individual components of an instruction.

Grammar for the SMM register transfer language

```
<program>             ⟶   (<instruction>*)
<instruction>         ⟶   (<label> <delabeled-instr>) | (<delabeled-instr>)
<delabeled-instr>     ⟶   <opcode-operands> | <opcode-operands> <if-clause>
<opcode-operands>     ⟶   A <dest> <source> | L <dest> <source> |
                          DA <dest> <source> | DL <dest> <source> |
                          NOP | SIM <opcode-operands> <opcode-operands> |
                          SET-WAIT-BIT | CLEAR-WAIT-BIT |
                          ALU-LOAD-MODE <load-mode-value> |
                          MUL-LOAD-MODE <load-mode-value> |
                          JUMP <next-address> | WAIT
<dest>                ⟶   <reference-1>
<source>              ⟶   <reference-1> | <math-clause>
<reference-1>         ⟶   <reference> | (LATCH) | (NONE)
<reference-2>         ⟶   <reference> | (NONE)
<reference>           ⟶   <reg-ref> | <sram-ref>
<reg-ref>             ⟶   (R <regnum>) | (REG <regnum>) | (REGISTER <regnum>)
<regnum>              ∈   {x|0 ≤ x ≤ 63, x ∈ N}
<sram-ref>            ⟶   (S <sramaddr>) | (SRAM <sramaddr>)
<sramaddr>            ∈   {x|0 ≤ x ≤ 4091, x ∈ N}
<math-clause>         ⟶   (<math-op> <reference-1> <reference-2>)
<math-op>             ⟶   + | * | -* | - | CMP | CMP0 | I | F2I | I2F | F2IS
<next-address>        ⟶   <label> | <absolute-addr>
<absolute-addr>       ⟶   {x|0 ≤ x ≤ 2047, x ∈ N}
<label>               ⟶   :<label-char><label-num-char>*
<label-char>          ∈   [a-zA-Z]
<label-num-char>      ∈   [a-zA-Z0-9]
<load-mode-value>     ∈   {x|0 ≤ x ≤ 63, x ∈ N}
<if-clause>           ⟶   (LCC) | (LCC GOTO <next-address>) |
                          (GOTO <next-address>) |
                          (LCC IF <condition-code> <next-address>) |
                          (IF <condition-code> <next-address>)
<condition-code>      ⟶   < | > | cc0 | cc1 | cc2 | cc3 | cc4 | cc5 | cc6 | cc7
```

Most instructions are of the form:

```
(<opcode> <dest> <source> <if-clause>)
```

although not all instructions neceassarily contain all four elements.

C.1 Instruction Opcodes

The first element of every instruction is the <opcode>, an identifier for the type of operation requested. Each opcode is described briefly below:

C.1.1 NOP

The `NOP` instruction does nothing ("No-OPeration"). It occupies one SMM cycle. `NOP` instructions can have labels and <if-clauses> like other instructions.

C.1.2 ASSIGN

The `ASSIGN` instruction, denoted by an "A" or "L" opcode (L for "load") is the most common instruction in SMM RTL. It is used to move data from one place to another, possibly performing some function on it along the way. The <dest> of an `ASSIGN` is given the value of the <source>. If the <source> is a <math-clause>, the value of the math operation is calculated first, and that value is then assigned to the <dest>. Example:

```
(A (S 42) (+ (R 5) (R 10)))
```

Assign the sum of register 5 and register 10 to SRAM 42.

C.1.3 DOUBLE ASSIGN

The `DOUBLE ASSIGN` instructions "DA" and "DL" are macro instructions which permit easy manipulation of 64-bit double-precision numbers. A `DOUBLE ASSIGN` instruction is expanded at assembly time into two sequential `ASSIGN` instructions, with the source and destination locations "incremented" to reference the high-order word for the second ASSIGN. All arguments to a DOUBLE ASSIGN should be low-order words of double-precision numbers (assumed to be even registers or SRAM locations). Example:

```
(DA (S 8) (+ (R 4) (S 42)))
```

expands into the two instructions

```
(A (S 8) (D1+ (R 4) (S 42)))
(A (S 9) (D2+ (R 5) (S 43)))
```

The pseudo-operations "D1+" and "D2+" represent the two phases of a 64-bit add operation (it takes two cycles to completely load both operands).

Note: If a "DA" instruction has a label, the label will be associated with the first ASSIGN instruction. If a "DA" instruction has an (LCC) as part of it's <if-clause>, that latch will occur on the *first* ASSIGN instruction. The remainder of the <if-clause> will take place on the *second* instruction.

C.1.4 SIMULTANEOUS

The SIMULTANEOUS ("SIM") instruction allows the user to specify two orthogonal operations which should occur during the same cycle on SMM. For example, the user may wish to load a register with a value from SRAM while starting a computation in the ALU. Since these two operations do not share any portion of SMM hardware, they may be made SIMULTANEOUS. Example:

```
(SIM (A (R 4) (S 8)) (A (S 12) (+ (R 22) (R 26))))
```

Note: The SIMULTANEOUS instruction is single-precision only. No double-precision version of "SIM" currently exists.

C.1.5 SET-WAIT-BIT

The SET-WAIT-BIT instruction forces the default "wait bit" of successive generated instructions to be set (1). Otherwise this instruction is identical to the NOP instruction.

C.1.6 CLEAR-WAIT-BIT

The CLEAR-WAIT-BIT instruction forces the default "wait bit" of successive generated instructions to be cleared (0). Otherwise this instruction is identical to the NOP instruction.

C.1.7 ALU-LOAD-MODE

The ALU-LOAD-MODE instruction is used to effect "load mode" operations inside the ALU. The "load mode" functions deal with such operations as rounding mode and denormalized numbers. These functions are detailed in the Weitek specifications. The <load-mode-value> given in the **ALU-LOAD-MODE** instruction is sent as the load control field to the ALU.

C.1.8 MUL-LOAD-MODE

The MUL-LOAD-MODE instruction is identical to the **ALU-LOAD-MODE** instruction except that it deals with the multiplier instead of the ALU.

C.1.9 JUMP

The JUMP instruction causes execution to jump to the instruction at <next-address>. This instruction is functionally equivalent to a **(NOP (GOTO <next-adress>))** instruction.

C.1.10 WAIT

The WAIT instruction is a macro which expands into a five-instruction loop. Upon entering the loop, the WAIT signal is set (raised). Execution remains in the loop until SMM receives a DONE signal from the host, at which time the WAIT signal is cleared. WAIT allows easy interlocking between the host and SMM. A (WAIT) expands into the following five instructions (*n* is a counter incremented with each WAIT instruction):

```
(set-wait-bit)
(:*wait-n-start* nop (none) (none) (if cc4 :*wait-n-loop*))
(nop (goto :*wait-n-done*))
```

```
(:*wait-n-loop* nop (goto :*wait-n-start))
(:*wait-n-done clear-wait-bit)
```

C.2 Destinations

In most instructions, the second list element is the <dest>, or destination. The <dest> tells *chasm* what machine location is affected by the action of the instruction, such as where a computed value should be stored. There are four types of destinations.

C.2.1 Register Destinations

By far the most commonly used destination is the register file. The register file contains 64 memory locations, or registers. Generally, values are loaded from the SRAM into the register file at the beginning of a program, computation is carried out using the register file locations, and final results are written back to the SRAM.

A register file destination consists of a keyword (either R, REG, or REGISTER) and an address. Valid addresses range between 0 and 63. Thus, the destination

```
(REG 42)
```

represents address 42 in the register file.

C.2.2 SRAM Destinations

Sending data to the Static RAM is similar to using the register files. All SRAM destinations have a keyword (S or SRAM) and an address between 0 and 4091. To send data to SRAM location 17, simply use the destination

```
(SRAM 17)
```

C.2.3 The LATCH Destination

In order to facilitate quick reuse of computed values, SMM has a feedback path from the output of the ALU/multiplier back to the input port. This feedback path is known as the LATCH, and to use it, simply use the destination:

```
(LATCH)
```

By its nature, the LATCH does *not* have state. Data values which are sent to the LATCH must be referenced from the LATCH exactly when they are available on the bus (either 12 (or 10) cycles after initiating an ALU (or multiplier) operation). Failure to source the LATCH properly will cause whatever data which was sent to the LATCH to be lost. The assembler will attempt to issue warnings whenever it thinks data may be lost due to incorrect timing, but this function is not currently fully implemented.

C.2.4 The NONE Destination

Sometimes it is necessary to compute a value but not save the computed result, such as when performing comparisons. In these cases, data on the bus should not be saved anywhere. To indicate that the result of an instruction is irrelevant, use the (NONE) destination. The (NONE) destination guarantees that no device will read the bus and load the results of the instruction.

C.3 Sources

Just as most instructions need to know where to send their results, they also need to know from where to get their results. The <source> of an instruction contains this information. Sources may be simple or complex. Simple sources are exactly like destinations. They may be register file locations, SRAM locations, the LATCH, or NONE. Complex sources represent the result of a mathematical operation on two simple sources. All complex sources are of the form:

```
(<math-op> <a-port-source> <b-port-source>)
```

For example, the expression `(* (R 17) (R 42))` stands for the product of the values in register locations 17 and 42. Many valid math operations exist; more may be added as necessary.

Note: If the LATCH is to be used in a complex source, it *must* be listed as the A-port source. Due to the design of SMM, the contents of the LATCH may not be loaded into the B-port.

C.4 If Clauses

The fourth and final portion of each instruction is the <if-clause>. The <if-clause> contains all information necessary for computing where execution should proceed upon completion of the instruction. If clauses come in three major types: none, GOTO, and IF. In addition, the keyword `LCC` may be added to any if clause to indicate that SMM should latch the condition bits.

C.4.1 No If Clause Present

If an instruction does not have an if clause (or the if clause is simply the direction `(LCC)`, execution proceeds to the next sequential instruction.

C.4.2 The GOTO-type If Clause

An if clause of the form `(GOTO <address>)` or `(LCC GOTO <address>)` causes execution to unconditionally branch to the instruction at <address>. The specified <address> may be either an absolute numeric address or the label of another instruction.

C.4.3 The IF-type If Clause

An if clause of the form `(IF <condition> <address>)` or `(LCC IF <condition> <address>)` denotes a conditional branch. The <condition> should be a valid condition code identifier, representing one of the eight available condition bits. If the <condition> bit is set (high), then execution branches to the instruction at the specified <address>. If the <condition> is clear (low), then execution proceeds to the next sequential instruction. Again, the specified <address> may be either an absolute location of an instruction label.

C.5 Example Instructions

A few example instructions are show below:

```
;; Assigns result to multiple destinations.  Since the goto field has
;; been omitted, the program would proceed to the next instruction.

(da ((s saddr) (r raddr) (latch)) (oper (r raddr1) (r raddr2)))

;; First operand can be from the latch instead of register file.  The
;; condition bits of the result are latched by using lcc.

(da ((s saddr) (r raddr)) (oper (latch) (r raddr2)) (lcc))

;; Operation with one operand.  The condition bit cc1 is tested, and
;; if it is equal to one, a branch occurs to the instruction marked by
;; :foo.

(da (s saddr) (oper (latch)) (if cc1 :foo))

;; Condition bits of result are latched and the program goes to the
;; instruction marked by :do.

(da (r raddr) (oper (r raddr1) (r raddr2)) (lcc goto :do))

;; Equivalent to nop but allows use of branching field.  The location
;; none serves as a place holder for the assembler.  No actual
;; location is accessed.

(da (none) (none) (goto :loop))
```

D The SMM Software Suite

This portion of the memo describes the software tools available for the Standard Map Machine. Section D.1 describes the primitive functions added to Scheme which allow communication with SMM. In Section D.2 we describe some of the simple procedures which exist to facilitate downloading of code to SMM and bidirectional transfer of data. Section D.3 describes breifly how to run the assembler, and Section D.5 lists warning messages which might be issued during assembly.

D.1 New Scheme Primitives

Three primitive operations were added to the CScheme Microcode to allow Scheme processes to communicate with SMM. Acually, these primitives will work with any HP memory-mapped device, not just SMM. These primitives are defined in the file

```
zurich:/scheme/users/chaos/microcode/chaos.c
```

A brief description of each new primitive follows:

D.1.1 (init-memory-mapped-device *string*)

The `init-memory-mapped-device` primitive initializes a page of memory for communication with a memory-mapped device and tells the Unix kernel that accesses to this area of memory should be directed to a specific device. `init-memory-mapped-device` takes one argument, a *string* containing the full name of the device file to be accessed (we used "/dev/chaos" for SMM). The device file must already exist and be configured to point to the proper slot where the card resides. If `init-memory-mapped-device` succeeds, it returns the base address of the memory block assigned to the device (this value is also maintained internal to Scheme; we return it here to signify success of the initialization operation). If `init-memory-mapped-device` fails, it returns `#!FALSE`.

D.1.2 (write-memory-mapped-device! *address data-word*)

The `write-memory-mapped-device!` primitive takes as arguments an *address* and a *data-word*, both integer values, and "writes" the *data-word* at offset *address* in the memory-mapped device block of reserved memory. When the write to the reserved block of memory is attempted, it is converted into an operation to SMM. The address value is sent over the address lines to SMM, and the data value is sent over the data bus. As both busses are 16 bits wide, the binary representations of *address* and *data* cannot exceed 16 bits.

D.1.3 (read-memory-mapped-device *address*)

The **read-memory-mapped-device** primitive takes an integer *address* as its argument and returns the integer data values "stored" at offset *address* in the device memory. Again, the read operation is intercepted by the kernel and changed into a request to the actual device. SMM reads the *address* from the address bus and send back 16 bits of data on the data bus.

D.2 Scheme Utilities

Once the primitive operations described above were installed into the Scheme system, higher-level utilities dealing with SMM could be built. Before using any of these functions, the Scheme primitives must be initialized as follows:

```
(define init-memory-mapped-device
        (make-primitive-procedure 'init-memory-mapped-device))
(define write-memory-mapped-device!
        (make-primitive-procedure 'write-memory-mapped-device!))
(define read-memory-mapped-device
        (make-primitive-procedure 'read-memory-mapped-device))
(init-memory-mapped-device "/dev/chaos")
```

Once the primitives have been initialized, all of the procedures below may be used.

D.2.1 (stop-clock!)

The `(stop-clock!)` procedure halts the internal SMM clock. The clock should always be stopped before performing any SMM memory operations from Scheme.

D.2.2 (start-board *value*)

The `start-board` procedure sends *value* through the high-order (bits 80-95) instruction latch to the high-order instruction register. The current values in the other instruction latches (bits 0-79) are sent to bits 0-79 of the instruction registers. The **start-board** procedure then starts the SMM system clock. Usually, **start-board** is called with *value* equal to zero.

D.2.3 (download-code *instr-vector*)

The `download-code` procedure takes as an argument a vector of instruction bits to be downloaded sequentially to SMM starting at instruction address 0. The `download-code` procedure is generally used in conjunction with *chasm-bits*, the output of the assembler.

D.2.4 (download-data *data-list*)

The `download-data` procedure is used to send constant vales to the SMM SRAM. The *data-list* is a list of one- or two-element lists. A two-element list is interpreted as an (*address data*) pair, with value *data* stored at locations *address* and *address*+1 (all *data* values are coerced to double-precision floating point values). A one-element list (*data*) instructs `download-data` to store the value *data* in memory immediately after the previous store. For example, the list

```
((0 1) (4 7.5) (12))
```

will cause value 1 to be stored at SRAM locations 0 and 1, value 7.5 to be stored at locations 4 and 5, and values 12 to be stored at locations 6 and 7.

D.2.5 (upload-data *memory-address*)

The `upload-data` precedure returns the floating-point value stored at SRAM locations (*memory-address*,*memory-address*+1) in the SMM.

D.2.6 (wait)

The `wait` procedure reads the status of the SMM "wait" bit. If the value returned by `(wait)` is even, then SMM is in a wait state and the host may access data memory.

D.2.7 (done)

The `done` procedure tells SMM that the host computer has finished accessing data memory and to continue processing.

D.3 Using *Chasm*

This section details the Scheme procedures provided to access SMM assembler.

D.3.1 (chasm-source <source>)

The `chasm-source` procedure takes one argument, <source>, and runs *chasm* using <source> as the program source code. <source> should be a list of instructions. `chasm-source` displays a message as each phase of assembly begins.

D.3.2 (chasm-file <filename>)

The `chasm-file` procedure assembles a single file of SMM source code. It takes one argument, the name of the file containing the source code (without the ".scm" extension). `chasm-file` reads the source file (adding the ".scm" extension), runs `chasm-source` over the source code, and dumps the resulting *chasm-bits* to <filename>.asm. Note: as `chasm-file` uses the Scheme `load` procedure to read the source file, the source code should be the last expression in the file.

D.3.3 (chasm-file-load <filename>)

This procedure is identical to the `chasm-file` procedure, except that it also performs a `(download-code *chasm-bits*)` after the compilation is completed.

D.3.4 (chasm-file-list <filename>)

This procedure is identical to the `chasm-file` procedure, except it also generates a listing file <filename>.lst. The listing file shows for each macro-expanded instruction the address, instruction label, source code, and generated bits.

D.4 Assembler Caveats and Programming Hints

Although *chasm* does a good job of converting RTL into SMM microinstructions, it currently fails to warn the programmer of certain types of illegal programs. SMM programmers should be aware of the following program constraints inposed by the SMM architecture:

D.4.1 Multiplier/ALU Restrictions

D.4.1.1 Load Modes: The load mode for the multiplier and the ALU must be set at the beginning of the program. It may be changed later to change the rounding mode of the chips. The `ALU-LOAD-MODE` and `MUL-LOAD-MODE` macros facilitate setting these values.

D.4.1.2 Latency in the ALU and Multiplier: Results of multiplier operations are sent to the desired destination *ten* instruction cycles after the execution of the instruction. Results of ALU operations are written to the appropriate destinations *twelve* cycles after the execution of the instruction. References to destinations of multiplier/ALU operations *before* the 10/12 cycles have passed will use the old value of the location.

D.4.1.3 Multiplier Cycle Time: The pipelining requirements of the Weitek multiplier place restrictions on when subsequent double-precision multiplier operations may start. After a multiplier operation is started, subsequent multiplier operations may only be started two, six, and ten cycles after the initial operation[4].

D.4.1.4 ALU Cycle Time: ALU pipelining requirements also place restrictions on the number of cycles between subsequent ALU operations. If a second ALU operation is started before the first ALU operation has finished[5], the the operations must be separated by an even number of cycles.

D.4.1.5 Interleaving ALU and Multiplier Operations: When interleaving ALU and multiplier operations, there must be at least two instruction cycles between a multiplier operation and the previous ALU operation.

D.4.2 Conditional Branches

The `LCC` directive causes the SMM to latch the Weitek condition code bits resulting from the operation performed. When using `LCC` with a double-precision macro (`DA` or `DL`), the latch occurs on the *first* of the two single-precision instructions. The

[4]This restriction only applies to multiplier operations started within 12 cycles of the previous multiplier operation

[5]That is, there are fewer than 14 cycles between the two instructions.

condition codes may be tested using an "if" clause *twelve* cycles after the LCC has been performed[6].

The if clause tests the specified condition code bit and causes execution to jump to the specified instruction if the condition is true. If clauses in double-precision macros occur during the *second* of the two single-precision instructions[7].

Currently, only the ALU comparison instruction is provided by the assembler, although it is simple to add functionality. The comparison function, *cmp*, takes two operands and compares their values. To test if the first operand is less than the second, latch the condition code bits and test if cc0 is true. To test if the first operand is greater than the second, latch the condition code bits and test if cc1 is true.

D.4.3 Using the Feed-Through Latch

The feed-through latch is used for fast serial computations by allowing the result of a multiplier/ALU operation to be directly fed back into the multiplier or ALU as the A-port input. To use the LATCH destination with the multiplier, the LATCH must be used as a source exactly *ten* cycles later. When using the LATCH destination with the ALU, the LATCH must be used as a source exactly *twelve* cycles later.

D.4.4 Reading from Data Memory

When reading from data memory, the entire data memory address is specified in the instruction. Data cannot be read from a location in memory ten cycles after a multiplier operation which writes to that location, and twelve cycles after an ALU operation which writes to that location.

D.4.5 Writing to Data Memory

When writing to data memory, only the low four bits of the address (the *intra-page* address) are specified in the actual instruction. The page of memory written to is

[6]That is, there must be the equivalent of at least twelve NOP instructions between the LCC and the if clause.

[7]Notice that since if clauses is placed in the second single-precision instruction and LCC directives are placed in the first single-precision instruction, only ten instruction cycles are needed between double-precision instruction instead of the normal twelve cycles.

specified by the result of the last `F->IS` operation. Page addresses range from 0 to 255; write address locations for each page range from 0 to 15.

D.5 Assembler Error Messages

This section details the possible warning/error messages which might be generated by *chasm*.

- **Unknown Opcode: <op> inserting NOP instruction.**:

  The assembler read <op> as an instruction opcode, and <op> is not one of the valid instruction types. The instruction in question was replaced with a `(NOP)`.

- **Destination not a list: <dest> converting...**:

  The destination <dest> was not in list format. All valid destinations are lists. The assembler wrapped the input destination in a list and is attempting to proceed.

- **Tried to Increment Unknown Dest: <dest> ignoring...**:

- **Tried to Increment Unknown Source: <source> ignoring...**:

  In the process of macro-expanding a DA or DL instruction, the assembler read destination <dest> (source <source>), which it was not able to parse. The destination (source) was not incremented for the second single-precision instruction.

- **Source is not a list: <source> converting...**: The source <source> was not in list format. All valid sources are lists. The assembler wrapped the input source in a list and is attempting to proceed.

- **Incrementing odd REGISTER <num> in DP..."**:

- **Incrementing odd SRAM <num> in DP..."**:

  The source to a DA or DL instruction was an odd register or SRAM location. These instructions take the low-order word of a double-precision number, which by convention is in a even location.

- **Unmatched use of LATCH, no reference beyond end of code.**:

  The `(LATCH)` was used as a destination, but the code vector ends before that value is available.

- **Unmatched use of LATCH, no reference in instr <instr> number <future-num>:**

  The (LATCH) was used as a destination, but no matching source reference was made 12 (for ALU) or 10 (for multiplier) cycles later.

- **Warning: last placed instr not null...:**

  The last instruction in the vector, normally clobbered into a (JUMP 0) instruction to start the machine, was not a (NOP). This should not happen, ever.

- **Unknown Label <label> assuming instruction 0:**

  A label was referenced, but no instruction was tagged with it in the source program. The assembler has assumed address 0 as the address of the label.

- **Source length too long: <source> ignoring extraneous stuff:**

- **Dest length too long: <dest> ingoring extraneous stuff:**

  An unusually long (more than two element) source (destination) was read. Everything after the first two elements was ignored.

- **Illegal Port: <port> assuming A-PORT:**

  An attempt was made to reference the LATCH from a port other than the A-PORT. The LATCH is only connected to the A-PORT.

- **Unknown Source: <source> assuming LATCH:**

- **Unknown Dest: <dest> assuming LATCH:**

  A one-element <source> (or <dest>) was read, but it was neither (NONE) nor (LATCH). The assembler has assumed it was a (LATCH) and continued.

- **Unknown Source: <source> assuming REGISTER:**

- **Unknown Dest: <dest> assuming REGISTER:**

  A two-element <source> (<dest>) was read, but it was neither a register reference nor an SRAM access. The assembler has assumed it was a register reference.

- **Unknown MathOp: <mathop> assuming +:**

  The assembler read a math operator which it didn't recognize as valid. The unknown operator was replaced with +.

- **Generating bits for unknown port: <port> ignoring**:

  In the process of generating the actual bits for an instruction, the assembler was told to use port <port>, which it didn't recognize as valid.

- **Unknown CC: <cc> assuming code 0**

  The unknown condition code <cc> was read in an if-clause. The code was replaced with cc0.

References

[1] Abelson, H. and Sussman, G. J. with Sussman, J., *Structure and Interpretation of Computer Programs.* M.I.T. Press, MA, 1984.

[2] Applegate, J., et. al., *A Digital Orrery.* IEEE Transactions on Computers, Vol. c-34, No. 9, September 1985.

[3] Chirikov, Boris V., *A Universal Instability of Many-Dimensional Oscillator Systems.* Physics Report 52, No. 5 (1979), pp. 263-379.

[4] Clinger, W. and Rees, J., ed., *The Revised^3 Report on the Algorithmic Language Scheme.* AI Memo 848a, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, September 1986.

[5] *HP Series 200 Accessory Development Guide.* Hewlett-Packard Co., Colorado, 1983.

[6] Higgs, M., *Advanced Schottky Load Management.* Texas Instruments Inc., 1987.

[7] Nieh, J., *Using Special-Purpose Computing to Examine Chaotic Behavior in Nonlinear Mappings.* S. B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, MA, May, 1989.

[8] Roylance, G. L., *Expressing Mathematical Subroutines Constructively.* AI Memo 999, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, November 1987.

[9] Sussman, G. J. and Wisdom, J., *Numerical Evidence that the Motion of Pluto is Chaotic.* AI Memo 1039, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, April 27, 1988.